NFP 121 Programmation avancée

En tant que lead developer Client chez IsCool Entertainment, au sein de l'équipe "Multi-joueur", j'ai la responsabilité du bon fonctionnement de la partie client de nos jeux, de leur architecture, ainsi que de la mise en place des outils permettant de facilité ou de rationnaliser le travail des autres développeurs. L'équipe de développeurs au sein de laquelle j'évolue comprend 5 personnes, 2 développeurs "client" Actionscript, dont moi, 1 développeur JAVA qui à la charge de la partie "game server", qui comprend le gameplay des jeux, et 2 développeurs .NET qui s'occupent de la partie Web et portail. Nos jeux sont des applications Facebook. La partie .NET se permet d'afficher la page avec un client flash et gère l'ensemble de la partie dite "sociale". Le client Flash est connecté de manière persistante au game server (par socket). Nous avons à l'heure actuelle deux jeux en production, que nous faisons évoluer de manière itérative: un jeu de Belote et un jeu proche d'un scrabble. Les joueurs peuvent effectuer des parties à plusieurs, d'où l'importance de maintenir une connexion persistante, et d'avoir une optimisation tant dans l'affichage que dans la gestion du gameplay - que ce soit côté client que serveur - afin de ne pas nuire à la fluidité du jeu.

# 1/ Le développement client

J'arrive dans l'équipe Multi-joueur fin 2010. Le développement de la partie client est balbutiant. L'écosystème Flash permet d'écrire des morceaux de script au milieu de la conception graphique, ce qui nuit fortement à sa compréhension. Cependant, ce fût le choix pris par les développeurs me précédant, en raison de leurs propres habitudes. Qui plus est, la livraison est totalement manuelle. Le "release manager" compile le programme sur son poste de travail, et le mettant en ligne par un copier/coller sur les serveurs de fichiers des environnements de tests ou de production. La programmation se fait à la demande, en "fixant" des morceaux de code, quitte à rendre opaque l'architecture globale du client.

En 2010, nous n'avons qu'un seul jeu: Belote. Le deuxième, Wordox, viendra bien plus tard, sera développé entre décembre 2011 et avril 2012, et bénéficiera de l'expérience acquise par l'équipe sur Belote

## 1.1/ Mise en place de la compilation automatisée

La première action que je mène est l'automatisation de la compilation du client. En effet, on ne peut pas considérer qu'un logiciel compilé sur le poste d'un développeur - qui est forcément configuré particulièrement - soit une opération saine. Selon les différents postes, un programme risque de fonctionner ou pas. L'exemple bête est la présence de lien vers une ressource (images, scripts, ...) dont l'emplacement est fourni dans le code "en dur". Lancé sur le poste d'un développeur, cela fonctionnera. Sur un autre poste, si la ressource n'est pas disponible, cela deviendra source de bug, voir le programme ne pourra plus compiler.

La compilation automatisée est le premier pas vers la mise en place d'un serveur de compilation ou serveur d'intégration continue. Ce dernier sera un environnement neutre considéré comme la référence de compilation. Si le serveur de compilation "s'allume en rouge", le développeur reprend son code immédiatement. De plus, ce serveur de compilation sera le support pour la mise en place des tests à automatiser: tests unitaires, validation de l'intégrité de code, analyse de la couverture du code par les tests, voir déploiement.

Pour automatiser le processus de compilation, je fais appel à un outil issu du monde JAVA, nommé "Maven". A l'aide d'un plugin ("FlexMojo"), on peut compiler du code Actionscript en utilisant le support de cycle de projet offert par Maven. (Maven offre des moyen simple de valider les dépendances à d'autres bibliothèques, d'effectuer la compilation, les tests unitaires, de livrer les artefacts de compilation, ainsi que fournir une analyse du programme).

Une fois le client compilable par Maven, et non plus par Flash, la mise en place du serveur est simple. Il s'agit à cette étape de paramétrage. Comme premier choix de serveur de compilation, je choisi Hudson, celui-ci étant gratuit. Plus tard, nous basculerons vers une solution payante mais offrant plus de fonctionnalités, à savoir Teamcity.

## 1.2/ Refonte de l'architecture

### 1.2.1/ Utilisation de l'UML pour client

Le code pré-existant du client est très difficile à conserver. Il est entremêlé, Les dépendances fortes étaient omniprésentes. Lorsqu'il faut permettre au client de supporter une fonctionnalité de connexion particulière, la modification d'une classe entraine la modification de l'ensemble du code. A ce stade-là, il ne me reste plus énormément de choix: en discussion avec le responsable technique de l'équipe, nous décidons de réécrire entièrement le client, afin de poser une architecture plus évolutive, où modifier un objet ou une classe ne force pas à changer de tout le programme.

Il n'est cependant pas question pour le responsable technique de me confier ce chantier - qui est coûteux - sans avoir en amont effectué un minimum d'analyse. Je choisis de baser celle-ci selon deux axes:

* Premièrement, je dois valider la thèse selon laquelle le code actuel était problématique
* Deuxième, je pose la base d'une architecture qui puisse être compréhensible par les autres membres techniques de l'équipe, rapidement.

Pour le premier point, j'utilise simplement un outil de retro-conception: celui-ci génère les diagrammes UML de classes à partir d'un code déjà existant. Le résultat est sans appel. Pour en revenir à une autre UE (MOCA B1), le diagramme de classes est un graphe pratiquement fortement connexe. Toutes les classes finissent par être reliées à toutes les autres. La notion de modularité est inexistante. L'absence d'interface de programmation transforme chaque modification en risque pour toute l'application. En ce basant sur des classes concrètes, toutes classes est fortement dépendantes de l'implémentation même de toute autre classe. Pire que le reste, les messages échangés avec le serveur sont traités "inline", un peu à tous les endroits du code, ce qui m'empêche de savoir simplement où et pourquoi le client de jeu réagit de telle ou telle manière après un échange avec le serveur. En effet, la bibliothèque utilisée pour communiquer avec le serveur (lib client de SmartFox) utilise les événements de Flash pour la partie communication - ce qui est totalement justifié. Cependant, en Actionscript, le type d'un événement est une simple chaîne de caractères. Je me retrouve donc avec un peu partout des références à ces chaînes de caractères, réécrites à même le code à chaque fois.

Je n'ai pas besoin de développer plus mes arguments pour mettre en valeur l'état du code. Qui plus est, l'utilisation d'outils de retro conception et d'UML est suffisamment rare dans l'équipe pour que cela donne plus de crédit à ma thèse.

Pour le deuxième point, j'utilise deux types de diagrammes UML. De séquences d'abord. Avant de penser au contenu même des classes, il me faut poser le concept d'un mécanisme de communication entre l'utilisateur de le game server, puisque le client n'est jamais qu'une interface assez évoluée entre les deux. Ainsi, pour les phases de connexions, d'entrée dans une partie, de jeu, de déconnexions, les diagrammes de séquences ont été définis. Ceux-ci font apparaître comme objets l'utilisateur, le game serveur, et entre eux:

* La "vue", à savoir toutes les objets d'affichage du client
* La partie contrôleur, celle qui agit sur les données du client, sur les vues, et sur la partie network du client
* Le modèle, où l'on trouve les objets stockant les données et états du client
* La couche network, qui interagit avec le serveur.

De ces diagrammes de séquences, on arrive facilement à ressortir les fonctionnalités dont on a besoin pour les différents blocs du client. Je peux ensuite les diviser en interfaces de classes.

Je ne pose pas le diagramme de classes complet du client de jeu. Les éléments qui m'intéressent le plus sont la partie contrôleur et la partie network. (Plus tard, je comprendrais que j'aurais dû tout de même poser les diagrammes du modèle. En effet, 4 développeurs différents vont passer sur le code, et l'absence de conception suffisamment élaborée en amont sur la partie modèle engendrera un assez gros problème. Si au démarrage, le modèle peut être représenté par deux ou trois classes, rapidement, son code deviendra impossible à relire, les classes gonfleront. Plus de design en amont aurait pu éviter ce problème, du moins, j'éviterais que cela ne se reproduise sur Wordox).

Les différents diagrammes posés, il n'est plus très difficile de démontrer la raison d'être du chantier de réécriture. Si celui-ci représente un coût au démarrage, l'investissement est plus que raisonnable.

J'admets ne pas être un grand utilisateur de diagrammes UML de classes. Je leur préfère largement:

* les diagrammes de séquences, parce qu'ils font bien plus facilement apparaître tous les composants d'un logiciel, ce qui est extrêmement important de notre architecture ou les composants sont nombreux et ne partagent pas la même technologie (Actionscript, JAVA, .NET, SQL). Nous avons pu par exemple corriger et éviter des monter en charge inutile sur les serveurs, en visualisant rapidement la manière dont le client requête la partie serveur. Ou encore, isoler les données que nous voulions stocker en cache (lorsqu'il y a bien plus de lecture de la donnée que d'écriture) ou en base de données (lorsque la donnée est aussi souvent écrite que lue, voir plus écrite que lue). A chaque nouvelle fonctionnalité touchant l'ensemble des composants, tous les développeurs se réunissent, et nous discutons ensemble autour de ces diagrammes, quelque soit notre technologie, ce qui permet à tous d'avoir une vision d'ensemble,
* les diagrammes d'états, parce qu'ils permettent de savoir rapidement qu'à un instant T, le client doit pouvoir recevoir tel message (que ce soit d'un serveur ou de l'utilisateur) ou effectuer telle action.

### 1.2.2/ Mise en place des tests unitaires, de l'analyse du code et de l'injection de dépendances. Implémentation du client

Depuis plusieurs années, afin de valider rapidement les différentes parties du code, j'utilise et je valorise l'utilisation de tests unitaires au sein des équipes dans lesquelles je travaille. A court terme, j'y trouve deux bénéfices directs:

* le temps passé en débogage est moins long, et moins pénible
* l'écriture des tests unitaires force une certaine structure du code (peu d'utilisation de singleton, suppression de classes statiques, injection des dépendances par le biais d'interfaces et on d'implémentations concrètes, ...)

Au sein du serveur d'intégration continue (Hudson), je rajoute donc la compilation des tests, l'analyse de la couverture par les tests, ainsi que des rapports de "conformité du code" selon différentes conventions de codages. Celles-ci sont trop nombreuses pour être énumérées, et vont de la détection de code inutilisé (dit mort) à l'analyse de la complexité cyclomatique du code (boucles, conditions, ...).

Je commence alors à écrire le code du client, en me basant sur les diagrammes de séquences et de classes posées plus tôt. Je n'ai pas encore la discipline nécessaire pour appliquer le TDD (Test Driven Development), qui consiste à écrire les tests unitaires avant l'écriture même du code dit "de production". Cependant, je m'offre des pauses systématiques, au moins une tous les jours, pendant lesquelles j'écris quelques tests sous forme d'une validation de mes méthodes par une forme de contrat (cf: PJ, Présentation des tests unitaires).

Pour ne pas tomber dans le même piège que mes prédécesseurs, je développerais en utilisant plusieurs concepts que j'ai acquis dans d'autres expériences.

* Tout d'abord, se baser sur des interfaces qui définissent un comportement, un contrat pour les classes les implémentant. Il y a de multiples intérêts à cela. Celui qui m'intéresse le plus est la fiabilité de la partie générique.
  + En effet, lorsque je dois écrire des tests unitaires pour une méthode ayant différentes dépendances, je ne dois pas utiliser les classes représentant ces dépendances comme le fait logiquement le client de jeu. Si je fais cela, mon test ne valide plus l'unité de code qu'est la fonction, mais valide la fonction ainsi que le fonctionnement de toutes les dépendances. Si jamais mon test rencontre un problème, je vais être incapable de savoir si mon erreur d'implémentation se trouve dans la méthode testée ou dans l'une des dépendances. Il va me falloir créer une fausse implémentation pour chaque dépendance, dont je définis à l'avance le fonctionnement. (On appellera cela un stub, voir un mock lorsqu'en plus de simuler une implémentation on ajoutera une logique de validation à même la dépendance). Ce mécanisme est plus simple à mettre en œuvre avec des interfaces qu'avec des classes concrètes dont on hérite.
  + Une deuxième raison est que je ne souhaite pas avoir à penser aux implémentations des dépendances. Je souhaite juste que, pour chaque dépendance qu'une classe a, celle-ci corresponde à un contrat. Typiquement, la couche de communication du client est capable de communiquer avec différents supports: le javascript (JS) contenu dans la page, ou le game server (socket), ou des webservices. Le code est articulé autour d'un gestionnaire auquel on ajoute des "frames", qui sont des blocs fonctionnels où seront traités un certain nombre de messages - aussi bien en réception qu'en envoie. Chaque frame est configuré pour un destinaire en particulier (JS, socket ou webservice). Lorsque le gestionnaire ajoute une frame, il lui fourni un objet correspondant à l'interface de communication dédiée. Il y a trois implémentations différentes de ces interfaces de communication, qui ont leur propre comportement et plusieurs méthodes qui sont liées au besoin à l'ouverture d'une connexion, à la gestion des erreurs de communication et autres. La frame n'a pas besoin de savoir que son interface de communication est capable de tout cela. Elle a uniquement besoin d'envoyer des messages, et d'écouter les messages la concernant provenant de l'extérieur. Cela se résume à deux méthodes.

En terme d'architecture, c'est extrêmement fluide. On peut câbler, recâbler n'importe quelle classe implémentant la bonne interface. De plus on limite la visibilité à ce qui est vraiment nécessaire, ce qui a le mérite de rendre le code plus compréhensible pour les autres développeurs, et évite que des méthodes non souhaitées soient appelées, parce qu'elles appartenaient à une implémentation concrète.

* Deuxième point, je mets en place un socle complet pour gérer toutes la partie réseau, en mettant un point d'honneur à fiabiliser et stabiliser toute la basse couche du client, au détriment si le temps me manque de la partie graphique. Certes, Flash n'est pas connu pour être la plateforme de développement la plus fiable, certes, l'Actionscript est uniquement un langage de script (ECMAScript), certes on peut faire un peu tout et n'importe quoi en Flash, et certes "Flash ça rame". Pour certains membres de l'équipe qui avant mon arrivée, considéraient que la partie Flash était "un asset graphique comme un autre", je me focalise trop sur des problématiques techniques, alors que le point fort de Flash est toute sa partie graphique. Je partage ce dernier avis, et c'est la raison pour laquelle je choisis pourtant de mettre le plus d'effort possible sur ce qui, in fine, est déterminant pour le succès de nos jeux: la stabilité de la connexion et de la jouabilité.

Je reprends l'ensemble des messages envoyés par le serveur ou le client, un par un. Aucune documentation des messages n'existe: nous l'écrivons avec le développeur JAVA. Pour chaque message, je choisis d'écrire une classe, qui me permettra de retrouver facilement tous les endroits du code où il apparaît. Chaque classe contient le code de sérialisation et désérialisation du contenu du message, ce qui amène à beaucoup de classes très simples, mais fastidieuses à écrire. J'en profite du coup pour poser les bases d'un générateur de protocole client / serveur, qui deux ans plus tard, sera la bible du fonctionnement client/server de nos jeux. La définition des messages est placée dans un fichier XML que le développeur Java et moi complétons. Le code est généré aussi bien en Java qu'en Actionscript, compilé et immédiatement livré sur un serveur "d'artefact Maven" (Nexus) que j'ai commandé à l'équipe infrasctruture.

Au final, ce chantier me permet de passer moins de temps à chercher à se trouve tel ou tel exécution, et me permet de me focaliser sur l'implémentation des fonctionnalités, tout en limitant l'ajout de bugs.

### 1.2.3/ Refactoring, extraction de comportements, lien avec des design patterns et autre patterns de conception

Tout au long du développement, je "refactore". J'entends par là:

* tomber sur un morceau de code qui à force de modification, commence à être difficile à comprendre, sembler étrange, donner un "bad smell"
* encadrer la fonctionnalité si besoin est pour m'assurer que je ne modifierai pas le résultat du code en modifiant son écriture
* modifier le code, très localement, pour l'améliorer.

Je ne me lance rarement, voir pas, dans ce que certains développeurs appelleront "gros refacto", et qui les mènent à tirer le code pendant des semaines dans un peu tous les sens. Non pas que je trouve cette dernière démarche forcément mauvaise, mais je n'appelle pas cela "refactorer", parce qu'à reprendre tout ou presque tout le code, il est difficile d'assurer qu'on n'a pas modifié le fonctionnement de l'application. Et, par ailleurs, je suis mauvais, ou mal à l'aise, dans les gros chantiers qui nous mette la tête sous l'eau pendant des semaines.

Cependant, je "refactore" à petite dose tout au long de l'écriture du programme. En effet, l'ajout de code se faisant au fil de l'eau, et n'ayant pas d'architecte dont le rôle est uniquement de veiller à l'ensemble des classes, il m'arrive de dupliquer un comportement sans le remarquer. C'est très fréquent des que l'on développe à plusieurs. Aussi, je trouve important de suivre et corriger cela de la même manière: au fil de l'eau.

Pour se faire, une connaissance de quelques design patterns est très utile, si ce n'est indispensable.

* Utiliser une fabrique (Pattern Factory) dès que plusieurs classes, implémentant la même interface, sont instanciées à différents endroits, afin d'assurer la souplesse de la généricité
* Extraire des comportements et regrouper plusieurs méthodes derrière un seul appel, dès qu'une classe un peu générique se voit peupler de méthodes qui ne sont jamais appelées selon la manière dont s'est déroulée l'instanciation (Pattern Strategy)

Concernant d'autres patterns, la connaissance de certains est inhérente au développement Flash, qui demande à bien en comprendre les concepts, comme le pattern Observer. Quand à d'autres comme le pattern Command et Mediator, ils sont omniprésents dans le Framework MVC Robotlegs, qui j'ai choisi d'adopter comme support pour le code du client.

### 1.2.4/ Mise en place d'un framework MVC orienté injection automatisée

L'implémentation du client bouge rapidement. Un peu trop et, étant donné que je n'ai pas totalement cadré la gestion du principe "Modèle Vue Contrôleur", nous nous retrouvons - avec l'autre développeur client - avec une structure de plus en plus incompréhensible dès que nous approchons du code gérant l'affichage du client. En soi, cela m'inquiète peu, il existe de nombreuses solutions pour remédier à cela, notamment l'adoption d'un "framework MVC", qui aura pour effet immédiat de structurer le code autour de son implémentation. Evidemment, on peut aussi en développer un par nous-mêmes. Cela prendrait plus de temps, mais sera plus satisfaisant.

Entre développeurs et lead developper Flash, nous commençons à étudier les différentes pistes. Il en existe plusieurs, dont certaines mènent à de véritables mastodontes architecturaux. Si, au démarrage, je décide de rester en retrait sur le sujet, ne voyant pas la situation se débloquer au bout de plusieurs semaines de tests, je choisis de trancher pour la partie Multijoueur. Nous utiliserons Robotlegs, sur le motif qu'il soit simple à mettre en œuvre, peut intrusif - même s'il apportera la structure que je lui demande - et est largement supporté par une communauté de développeurs.

La mise en œuvre de robotlegs est un chantier de moyenne envergure: seul, je me vois travailler dessus pendant 2 ou 3 semaines au plus, tout en m’offrant le luxe d’une intégration de plus en plus continue : aidé par le serveur d’intégration, je vais pouvoir vérifier et valider au jour le jour avec le responsable fonctionnel que le client répond toujours correctement au besoin. Qui plus est, si le code concernant l'affichage est devenu "hirsute", ce n'est pas le cas du reste. Les frames deviendront des "services Robotlegs", le code de traitement des messages qu'elles contiennent sera systématiquement déporté dans des commandes distinctes. Les classes de modèles deviendront des "acteurs Robotlegs", sans avoir à en modifier les codes.

La partie affichage (vue) par contre, va devoir subir une bonne refonte. Jusqu'à présent, je n'ai pas testé le code des vues, parce que celui-ci change trop souvent, pour un pixel à gauche, une couleur à droite, et qu'il est difficile de voir une rationalité à tester unitairement cette zone de code. Cependant, à l'aide de Robotlegs, je vais diviser toutes les classes de vue en deux composants:

* La vue, en tant que telle, qui ne gère vraiment que l'affichage et les interactions de l’utilisateur - mais plus la gestion des événements provenant des modèles.
* Le médiateur - qui pourra être lié d'ailleurs à plusieurs classes de vues - dont l'unique propos est d'être un intermédiaire entre les événements utilisateurs - qui seront renvoyés aux commandes - et les événements internes au client - modifications de modèle ou d'état, qui seront envoyées à la vue.

Dans ce cadre, je peux ajouter les tests sur les médiateurs, et augmenter par ce biais la zone de confiance que je peux faire au code du client.

Robotlegs se base sur un framework d'injection automatisée des dépendances (swiftsuspender). Etant donné que le code du client se base sur l'injection de dépendance, l'apport de cet outil automatisant le processus est énorme pour moi. Je ne suis plus obligé pour chaque classe de lui fournir ses dépendances à la main, lorsque pourtant, il s'agit toujours des mêmes objets instanciés à un endroit bien précis du code. Je choisis de déléguer cela à l'injecteur disponible, qui permet de "mapper" des objets avec des classes ou des interfaces. A l'exécution, lorsque l'injecteur trouvera une propriété précédée d'une metadonnée particulière, il lui fournira automatiquement l'objet correspondant au type souhaité, après l'avoir instancié où avoir réutilisé un objet déjà existant. Je vais perdre un peu en performance, mais cela ne sera pas vraiment visible: en effet, ce qui consomme le plus de "temps CPU" au sein du client reste l'affichage. Et, suite à la refactorisation ainsi qu'aux optimisations systématiques du code, l'exécution du client est fluide.

## 1.3/ A propos de la programmation événementielle

L'actionscript et la programmation événementielle sont très liés. Le moteur de Flash s'articule autour de sa "DisplayList". Le "stage" (composant visuel qui contiendra tous les autres) est une sorte d'animation. Pour déclencher un code, on utilisera le "EnterFrame" du stage par exemple, qui correspond à un rafraîchissement du visuel du flash. A ce niveau, le nombre d'événements EnterFrame disponible sera extrêmement dépendant du nombre d'images par seconde (FPS) que l'on est capable d'afficher. Un autre exemple: pour traiter un message venant d'un socket, on écoute l'événement ProgressData envoyé par le socket, qui contient un flux binaire.

Lorsqu'on développe en actionscript, on se lie systématiquement à un événement pour pouvoir effectuer un code, et on envoie d'autres événements pour pouvoir communiquer entre composants (le pattern Observer).

(A traiter :

2/ La mise en place des outils client/serveur

2.1/ Le générateur de protocole client/serveur

2.1/ Analyseur de statistiques client

)